Refinements for Free!

Cyril Cohen, Maxime Dénès, Anders Mörtberg

Refinements for free!*

Cyril Cohen[1], Maxime Dénès[2], and Anders Mörtberg[1]

[1] Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
{cyril.cohen,anders.mortberg}@gu.se
[2] INRIA Sophia Antipolis – Méditerranée
mail@maximedenes.fr

Abstract. Formal verification of algorithms often requires a choice between definitions that are easy to reason about and definitions that are computationally efficient. One way to reconcile both consists in adopting a high-level view when proving correctness and then refining stepwise down to an efficient low-level implementation. Some refinement steps are interesting, in the sense that they improve the algorithms involved, while others only express a switch from data representations geared towards proofs to more efficient ones geared towards computations. We relieve the user of these tedious refinements by introducing a framework where correctness is established in a proof-oriented context and automatically transported to computation-oriented data structures. Our design is general enough to encompass a variety of mathematical objects, such as rational numbers, polynomials and matrices over refinable structures. Moreover, the rich formalism of the Coq proof assistant enables us to develop this within Coq, without having to maintain an external tool.

Keywords: Coq, Data refinements, Formal proofs, Efficient algorithms and data structures, Parametricity

1 Introduction

It is commonly conceived that computationally well-behaved programs and data structures are more difficult to study formally than naive ones. Rich formalisms like the Calculus of Inductive Constructions, on which the Coq [6] proof assistant relies, allow for several different representations of the same mathematical object so that users can choose the one suiting their needs.

Even simple objects like natural numbers may have both a unary representation which features a very straightforward induction scheme and a binary one which is exponentially more compact, but usually entails more involved proofs. Their respective incarnations in the standard library of Coq are the two inductive types `nat` and `N` along with two isomorphisms `N.of_nat : nat -> N`

* The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

and `N.to_nat : N -> nat`. Recent versions of the library make use of ML-like modules and functors [4] to factor programs and proofs over these two types.

The traditional approach to abstraction is to first define an interface specifying operators and their properties, then instantiate it with concrete implementations of the operators with proofs that they satisfy the properties. However, this has at least two drawbacks in our context. First, it is not always obvious how to define the correct interface, and it is not clear if a suitable one even exists. Second, having abstract axioms goes against the type-theoretic view of objects with computational content, which means in practice that proof techniques like small scale reflection, as advocated by the SSReflect extension [9], are not applicable.

Instead, the approach we describe here consists in proving the correctness of programs on data structures designed for proofs — as opposed to an abstract signature — and then transporting them to more efficient implementations. We distinguish two notions: *program refinements* and *data refinements*. The first of these consists in transforming a program into a more efficient one computing the same thing using a different algorithm, but preserving the involved types. For example, standard matrix multiplication can be refined to a more efficient implementation like Strassen's fast matrix product [25]. The correctness of this kind of refinements is often straightforward to state. In many cases, it suffices to prove that the two algorithms are extensionally equal. The second notion of refinement consists in changing the data representation on which programs operate while preserving the algorithm, for example a multiplication algorithm on dense polynomials may be refined to an algorithm on sparse polynomials. This kind of refinement is more subtle to express as it involves transporting both programs and their correctness proofs to the new data representation.

The two kinds of refinements can be treated independently and in the following, we focus on data refinements. A key feature of these should be compositionality, meaning that we can combine multiple data refinements. For instance, given both a refinement from dense to sparse polynomials and a refinement from unary to binary integers we get a refinement from dense polynomials over unary integers to sparse polynomials over binary integers.

In a previous work [8], two of the authors defined a framework for refining *algebraic structures* in a comparable way, while allowing a step-by-step approach to prove the correctness of algorithms. The present work[3] improves several aspects by considering the following methodology:

1. *relate* a proof-oriented data representation with a more efficient one (Sect. 2),
2. *parametrize* algorithms and the data on which they operate by an abstract type and its basic operations (Sect. 3),
3. *instantiate* these algorithms with proof-oriented data types and their basic operations, and prove the correctness of that instance,
4. use *parametricity* of the algorithm (with respect to the data representation on which it operates), together with points 2 and 3, to deduce that the

[3] The formal development is available at `http://www.maximedenes.fr/coqeal/`

algorithm instantiated with the more efficient data representation is also correct (Sect. 4).

Further, this paper also contains a detailed example application of this new framework to Strassen's algorithm for efficient matrix multiplication (Sect. 5). Section 6 provides an overview of related work.

2 Data refinements

In this section we will study various data refinements by considering some examples. All of these fit in a general framework of data refinements based on heterogeneous relations which relate *proof-oriented* types for convenient proofs with *computation-oriented* types for efficient computation.

2.1 Refinement relations

In some cases we can define (possibly partial) functions from proof-oriented to computation-oriented types and *vice versa*. We call a function from proof-oriented to computation-oriented types an *implementation* function, and a function going the other way around a *specification* function.

Note that a specification function alone suffices to define a refinement relation between the two data types: a proof-oriented term p refines to a computation-oriented term c if the specification of c is p. We write the following helper functions to map respectively total and partial specification functions to the corresponding refinement relations:

```
Definition fun_hrel A B (f : B -> A) : A -> B -> Prop :=
  fun a b => f b = a.

Definition ofun_hrel A B (f : B -> option A) : A -> B -> Prop :=
  fun a b => f b = Some a.
```

Isomorphic types. Isomorphic types correspond to the simple case where the implementation and specification functions are inverse of each other.

The introduction mentions the two types `nat` and `N` which represent unary and binary natural numbers. These are isomorphic, which is witnessed by the implementation function `N.of_nat : nat -> N` and the specification function `N.to_nat : N -> nat`. Here, the proof-oriented type is `nat` and the computation-oriented type is `N`. Another example of isomorphic types is the efficient binary representation `Z` of integers in the Coq standard library that can be declared as a refinement of the unary, `nat`-based, representation `int` of integers in the SSReflect library.

Quotients. Quotients correspond to the case where the specification and implementation functions are total and where the specification is a left inverse of the implementation. This means that the computation-oriented type may have "more elements" and that the implementation function is not necessarily surjective (unless the quotient is trivial). In this case the proof-oriented type can be seen as a quotient of the computation-oriented type by an equivalence relation defined by the specification function, i.e. two computation-oriented objects are related if their specifications are equal. This way of relating types by quotients is linked to the general notion of quotient types in type theory [5]. The specification corresponds to the canonical surjection in the quotient, while the implementation corresponds to the choice of a canonical representative. However, here we are not interested in studying the proof-oriented type, which is the quotient type. Instead, we are interested in studying the computation-oriented type, which is the type being quotiented.

An important example of quotients is the type of polynomials. These are represented in SSReflect as a record type with a list and a proof that the last element is nonzero, however this proof is only interesting when developing theory about polynomials and not for computation. Hence a computation-oriented type can be just the list of coefficients and the specification function would normalize polynomials by removing zeros in the end.

A better representation of polynomials is sparse Horner normal form [10] which can be implemented as:

```
Inductive hpoly := Pc : A -> hpoly
                 | PX : A -> pos -> hpoly -> hpoly.
```

Here `A` is an arbitrary type and `pos` is the type of positive numbers, the first constructor represents a constant polynomial and `PX a n p` should be interpreted as $a + X^n p$ where a is a constant, n is a positive number and p is another polynomial in sparse Horner normal form. However, with this representation not only polynomials with zeros in the end can be represented but there are also multiple ways to represent polynomials like X^2 as it can be represented by either $0 + X^2 \cdot 1$ or $0 + X^1(0 + X^1 \cdot 1)$. To remedy this we implement a specification function that normalize polynomials and translate them to SSReflect polynomials.

Partial quotients. Quotient based refinement relations cover a larger class of data refinements than the relations defined by isomorphisms, but there are still interesting examples that are not covered, for example when the specification function is partial. To illustrate this, let us consider rational numbers. The SSReflect library contains a definition where they are defined as pairs of coprime integers with nonzero denominator:

```
Record rat : Set := Rat {
  valq : int * int;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}.
```

Here `‘|valq.1|` denotes the absolute value of the first projection of the `valq` pair. This definition is well-suited for proofs, notably because elements of type `rat` can be compared using Leibniz equality since they are normalized. But maintaining this invariant during computations is often too costly since it requires multiple gcd computations. Besides, the structure also contains a proof which is not interesting for computations but only for developing the theory of rational numbers.

In order to be able to compute efficiently we would like to refine this to pairs of integers (`int * int`) that are not necessarily normalized and perform all operations on the subset of pairs with nonzero second component. The link between the two representations is depicted in Fig. 1:

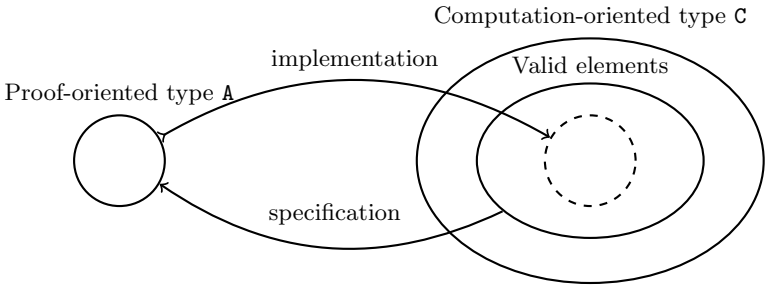

Fig. 1. Partial quotients

In the example of rational numbers the proof-oriented type is `rat` while the computation-oriented type is `int * int` and computations should be performed on the subset of valid elements of the computation-oriented type, i.e. pairs with nonzero second component. In order to conveniently implement this, the output type of the specification function has been extended to `option A` in order to make it total. The key property of the implementation and specification functions is still that the specification is a left inverse of the implementation. This means that the proof-oriented type can be seen as a quotient of the set of valid elements, i.e. elements that are not sent to `None` by the specification function. For rational numbers the implementation and specification functions and their correctness looks like:

```
Definition rat_to_Qint (r : rat) : int * int := valq r.
Definition Qint_to_rat (r : int * int) : option rat :=
  if r.2 != 0 then Some (r.1%:Q / r.2%:Q) else None.

Lemma Qrat_to_intK :
  forall (x : rat), Qint_to_rat (rat_to_Qint x) = Some x.
```

The notation `%:Q` is the cast from `int` to `rat`. Here the lemma says that the composition of the implementation with the specification is the identity. Using

this, we get a relation between `rat` and `int * int` by using `ofun_hrel` defined at the beginning of this section:

```
Definition Rrat : rat -> int * int -> Prop := ofun_hrel Qint_to_rat.
```

Functional relations. Partial quotients often work for the data types we define, but fails to describe refinement relations on functions. Given two relations `R : A -> B -> Prop` and `R' : A' -> B' -> Prop` we build a relation on the function space: `R ==> R' : (A -> A') -> (B -> B') -> Prop`. It is a heterogeneous generalization of the `respectful` functions defined for generalized rewriting [22].

This definition is such that two functions are related by `R ==> R'` if they send related inputs to related outputs. We can now use this to define the correctness of addition on rational numbers:

```
Lemma Rrat_addq : (Rrat ==> Rrat ==> Rrat) +rat +int*int.
```

The lemma states that if the two arguments are related by `Rrat` then the outputs are also related by `Rrat`.

However, we have left an issue aside: we refined `rat` to `int * int`, but this is not really what we want to do as the type `int` is itself proof-oriented. Thus, taking it as the basis for our computation-oriented refinement of `rat` would be inefficient. Instead, we would like to express that `rat` refines to `C * C` for *any* type `C` that refines `int`. The next section will explain how to program, *generically*, operations in the context of such parametrized refinements. Then, in Sect. 4, we will show that correctness can be proved in the specific case when `C` is `int`, and automatically transported to any other refinement by taking advantage of parametricity.

2.2 Comparison with the previous approach

We gain in generality with regard to the previous work on refinements [8] in several ways. The previous work assumed a total injective implementation function, which intuitively corresponds to a partial isomorphism: the proof-oriented type is isomorphic to a subtype of the computation-oriented type. Since we do not rely on those translation functions anymore, we can now express refinement relations on functions. Moreover, we take advantage of (possibly partial) specification functions, rather than implementation functions.

Another important improvement is that we do not need any notion of equality on the computation-oriented type anymore. Indeed, the development used to rely on Leibniz equality, which prevented us from using setoids [2] as computation-oriented types. In Sect. 2.1, we use the setoid `int * int` of rational numbers, but the setoid equality is left implicit. This is in accordance with our principle never to do proofs on computation-oriented types. We often implement algorithms to decide equality, but these are treated as any other operation (Sect. 3).

2.3 Indexing and using refinements

We use the Coq type class mechanism [23] to maintain a database of lemmas establishing refinement relations between proof-oriented and computation-oriented terms. The way this database is used is detailed in Sect. 4.

In order to achieve this, we define a heterogeneous generalization of the `Proper` relations from generalized rewriting [22]. We call this class of relations `param` and define it by:

```
Class param (R : A -> B -> Prop) (a : A) (b : B) := param_rel : R a b.
```

Here `R` is meant to be a refinement relation from `A` to `B`, and we can register an instance of this class whenever we have two elements `a` and `b` and a proof that `R a b`. For example, we register the lemma `Rrat_addq` from Sect. 2.1 using the following instance:

```
Instance Rrat_addq : param (Rrat ==> Rrat ==> Rrat) +rat +int*int.
```

Given a term x, type class resolution searches for y and a proof of `param` R x y. If R was obtained from a specification function, then $x =$ spec y and we can always substitute x by spec y and compute y, thus taking advantage of our framework to do efficient computation steps within proofs.

3 Generic programming

We may want to provide operations on the computation-oriented type corresponding to operations on the proof-oriented type. For example, we want to define an addition `addQ` on computation-oriented rationals `C * C`, corresponding to the addition `(+rat)` on `rat`. However this computation-oriented operation relies on both addition `(+C)` and multiplication `(*C)` on `C`, so we parametrize `addQ` by `(+C)` and `(*C)`:

```
Definition addQ C (+C) (*C) : (C * C) -> (C * C) -> (C * C) :=
  fun x y => (x.1 *C y.2 +C y.1 *C x.2, x.2 *C y.2).
```

This operation is correct if `(+rat)` refines to `(addQ C (+C) (*C))` whenever `(+int)` refines to `(+C)` and `(*int)` refines to `(*C)`. The refinement from `(+rat)` to `(addQ C (+C) (*C))` is explained in Sect. 4.1.

Since we abstracted over operations of the underlying data type, only one implementation of each algorithm suffices, the same code can be used for doing both correctness proofs and efficient computations as it can be instantiated by both proof-oriented and computation-oriented types and programs. This means that the programs need only be written once and code is never duplicated, which is an improvement compared to the previous development.

In order to ease the writing of this kind of programs and refinement statements in the code, we use operational type classes [24] for standard operations like addition and multiplication together with appropriate notations. This means we define a class for each operator and a generic notation referring to the corresponding operation. For example, in the code of `addQ` we can always write `(+)` and `(*)` and let the system infer the operations,

```
Instance addQ C `{add C, mul C} : add (C * C) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).
```

Here `‘{add C, mul C}` means that `C` comes with type classes for addition and multiplication operators. Declaring `addQ` as an instance of addition on `C * C` enables the use of the generic `(+)` notation to denote `addQ`.

4 Parametricity

The approach presented in the above section is incomplete though: once we have proven that the instantiation of a generic algorithm to proof-oriented structures is correct, how can we guarantee that other instances will be correct as well? Proving correctness directly on computation-oriented types is precisely what we are trying to avoid.

Informally, since our generic algorithms are polymorphic in their types and operators, their behavior has to be uniform across all instances. Hence, a correctness proof should be portable from one instance to another, so long as the operators instances are themselves correct.

The exact same idea is behind the interpretation of polymorphism in relational models of pure type systems [3]. The present section builds on this analogy to formalize the automated transport of a correctness proof from a proof-oriented instance to other instances of the same generic algorithm.

4.1 Splitting refinement relations

Let us illustrate the parametrization process by an example on rational numbers. For simplicity, we consider negation which is implemented by:

```
Instance oppQ C `{opp C} : opp (C * C) :=
  fun (a : Q C) => (-_C a.1, a.2).
```

The function takes a negation operation in the underlying type `C` and define negation on `C * C` by negating the first projection of the pair (the numerator). Now let us assume that `C` is a refinement of `int` for a relation `Rint : int -> C -> Prop` and that we have:

```
(Rint ==> Rint) (-_int) (-_C)
(Rrat ==> Rrat) (-_rat) (oppQ int (-_int))
```

The first of these states that the `(-_C)` parameter of `oppQ` is correctly instantiated, while the second one expresses that the proof-oriented instance of `oppQ` is correct. Assuming this, we want to show that `(-_rat)` refines all the way to `oppQ`, but instantiated with `C` and `(-_C)` instead of their proof-oriented counterparts (`int` and `(-_int)`).

In order to write this formally, we define the product and composition of relations as `R * S := fun x y => R x.1 y.1 /\ S x.2 y.2` and `R \o S := fun x y => exists z, R x z /\ S z y`. Using this we can define the relation `RratC : rat -> C * C -> Prop` as `RratC := Rrat \o (Rint * Rint)`. We want to show:

```
(RratC ==> RratC) (-_rat) (oppQ C (-_C))
```

A small automated procedure, relying on type class instance resolution, first splits this goal in two, following the composition `\o` in the definition of `RratC`:

```
(Rrat ==> Rrat) (-_rat) (oppQ int (-_int))
(Rint * Rint ==> Rint * Rint) (oppQ int (-_int)) (oppQ C (-_C))
```

The first of these is one of the assumptions while the second relates the results of the proof-oriented instance of `oppQ` to another instance. This is precisely where parametricity comes into play, as we will show in the next section.

4.2 Parametricity for refinements

While studying the semantics of polymorphism, Reynolds introduced a relational interpretation of types [19]. Parametricity [27] is a reformulation based on the fact that if a type has no free variable, its relational interpretation expresses a property shared by all terms of this type. This result extends to pure type systems [3] and provides a meta-level transformation $[\![\cdot]\!]$ defined inductively on terms and contexts. In the closed case, this transformation is such that if $\vdash A : B$, then $\vdash [\![A]\!] : [\![B]\!]\, A\, A$. That is, for any term A of type B, it gives a procedure to build a proof that A is related to itself for the relation interpreting the type B.

The observation we make is that the last statement of Sect. 4.1 is an instance of such a *free theorem*. More precisely, we know that $[\![\texttt{oppQ}]\!]$ is a proof of

$$[\![\forall \texttt{Z}, (\texttt{Z} \to \texttt{Z}) \to \texttt{Z} * \texttt{Z} \to \texttt{Z} * \texttt{Z}]\!]\ \texttt{oppQ}\ \texttt{oppQ}$$

which expands to

$$\begin{array}{lll} \forall \texttt{Z} : \texttt{Type}, & \forall \texttt{Z}' : \texttt{Type}, & \forall \texttt{Z}_\texttt{R} : \texttt{Z} \to \texttt{Z}' \to \texttt{Prop}, \\ \forall \texttt{oppZ} : \texttt{Z} \to \texttt{Z}, & \forall \texttt{oppZ}' : \texttt{Z}' \to \texttt{Z}', & [\![\texttt{Z} \to \texttt{Z}]\!]\ \texttt{oppZ}\ \texttt{oppZ}' \to \\ \multicolumn{3}{r}{[\![\texttt{Z} * \texttt{Z} \to \texttt{Z} * \texttt{Z}]\!] \quad (\texttt{oppQ}\ \texttt{Z}\ \texttt{oppZ}) \quad (\texttt{oppQ}\ \texttt{Z}'\ \texttt{oppZ}').} \end{array}$$

Then, instantiating `Z` to `int`, `Z'` to `C` and $\texttt{Z}_\texttt{R}$ to `Rint` gives us the exact statement we wanted to prove, since $[\![\texttt{Z} \to \texttt{Z}]\!]$ is what we denoted $\texttt{Z}_\texttt{R}$ `==>` $\texttt{Z}_\texttt{R}$.

Following the term transformation $[\![\cdot]\!]$, we design a logic program in order to derive proofs of closed instances of the parametricity theorem. Indeed, it should be possible in practice to establish the parametric relation between two terms like `oppQ` and itself, since `oppQ` is closed.

For now, we can only express and infer parametricity on polymorphic expressions (no dependent types allowed), by putting the polymorphic types outside the relation. Hence we do not need to introduce a quantification over relations.

4.3 Generating the parametricity lemma

Rather than giving the details of how we programmed the proof search using type classes and hints in the Coq system, we instead show an execution of this logic program on our simple example, starting from:

```
(Rrat ==> Rrat) (-rat) (oppQ C (-C))
```

Let us first introduce the variables and their relations, and we get to prove

```
(Rint * Rint) (oppQ int (-int) a) (oppQ C (-C) b)
```

knowing that `((Rint ==> Rint) (-int) (-C))` and `((Rint * Rint) a b)`. By unfolding `oppQ`, it suffices to show:

```
(Rint * Rint) (-int a.1, a.2) (-C b.1, b.2)
```

To show that, we use parametricity theorems for the pair constructor `pair` and eliminators `_.1` and `_.2`. In our context, we have to give manual proofs for them. Indeed, we lack automation for the axioms, but the number of combinators to treat by hand is negligible compared to the number of occurrences in user-defined operations. These lemmas look like:

```
param_pair := forall RA RB, (RA ==> RB ==> RA * RB) pair pair
param_fst := forall RA RB, (RA * RB ==> RA) _.1 _.1
param_snd := forall RA RB, (RA * RB ==> RB) _.2 _.2
```

Unfolding the last of these gives:

```
forall (RA : A -> A' -> Prop) (RB : B -> B' -> Prop)
       (a : A) (a' : A') (b : B) (b' : B'),
  RA a a' → RB b b' → (RA * RB) (a, b) (a', b')
```

This can be applied to the initial goal, giving two subgoals:

```
Rint (-int a.1) (-C b.1)
Rint a.2 b.2
```

The second of these follow directly from `param_snd` and to show the first it suffices to prove:

```
(Rint ==> Rint) (-int) (-C)
Rint a.1 b.1
```

The first of these is one of the assumptions we started with and the second follows directly from `param_fst`.

5 Example: Strassen's matrix product

In the previous development an important application of the refinement framework was Strassen's algorithm for the product of two square matrices of size n with time complexity $\mathcal{O}(n^{2.81})$ [25]. We show here how we adapted it to the new framework described in this paper.

Let us begin with one step of Strassen's algorithm: given a function f which computes the product of two matrices of size `p`, we define, generically, a function `Strassen_step f` which multiplies two matrices of size `p + p`:

```
Variable mxA : nat -> nat -> Type.

Context `{hadd mxA, hsub mxA, hmul mxA, hcast mxA, block mxA}.
Context `{ulsub mxA, ursub mxA, dlsub mxA, drsub mxA}.

Definition Strassen_step {p : positive} (A B : mxA (p+p) (p+p))
  (f : mxA p p -> mxA p p -> mxA p p) : mxA (p+p) (p+p) :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in let Y := B22 - B12 in
  let C21 := f X Y in let X := A21 + A22 in
  let Y := B12 - B11 in let C22 := f X Y in
  let X := X - A11 in let Y := B22 - Y in
  let C12 := f X Y in let X := A12 - X in
  let C11 := f X B22 in let X := f A11 B11 in
  let C12 := X + C12 in let C21 := C12 + C21 in
  let C12 := C12 + C22 in let C22 := C21 + C22 in
  let C12 := C12 + C11 in let Y := Y - B21 in
  let C11 := f A22 Y in let C21 := C21 - C11 in
  let C11 := f A12 B21 in let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

The `mxA` variable represents the type of matrices indexed by their sizes. The various operations on this type are abstracted over by operational type classes, as shown in Sect. 3. Playing with notations and scopes allows us to make this generic implementation look much like an equivalent one involving SSReflect matrices.

Note that `Strassen_step` expresses matrix sizes by the `positive` type. These are positive binary numbers, whose recursion scheme matches the one of Strassen's algorithm through matrix block decomposition. This is made compatible with the `nat`-indexed `mxA` type thanks to a hidden coercion `nat_of_pos`.

The full algorithm is expressed by induction over `positive`. However, in order to be able to state parametricity lemmas, we do not use the primitive `Fixpoint` construction. Instead, we use the recursion scheme attached to `positive`:

```
positive_rect : forall P : positive -> Type,
(forall p : positive, P p -> P (p~1)%positive) ->
(forall p : positive, P p -> P (p~0)%positive) ->
P 1%positive -> forall p : positive, P p
```

We thus implement three functions corresponding to the three cases given by the constructor of the `positive` inductive type: `Strassen_xI` for odd-sized matrices, `Strassen_xO` for even-sized ones and `Strassen_xH` for matrices of size 1. Strassen's algorithm is then defined as:

```
Definition Strassen :=
  (positive_rect (fun p => (mxA p p -> mxA p p -> mxA p p))
                 Strassen_xI Strassen_xO Strassen_xH).
```

Then we instantiate the `mxA` type and all the associated operational type classes to SSReflect proof-oriented matrix type and operators. In this context, we prove the program refinement from the naive matrix product `mulmx` to Strassen's algorithm:

```
Lemma StrassenP p : param (eq ==> eq ==> eq) mulmx (@Strassen p).
```

The proof is essentially unchanged from [8], the present work improving only the data refinement part. The last step consists in stating and proving the parametricity lemmas. This is done in a context abstracted over both a representation type for matrices and a refinement relation:

```
Context (A : ringType) (mxC : nat -> nat -> Type).
Context (RmxA : forall {m n}, 'M[A]_(m, n) -> mxC m n -> Prop).
```

Operations on matrices are also abstracted, but we require them to have an associated refinement lemma with respect to the corresponding operation on proof-oriented matrices. For instance, for addition we write as follows:

```
Context `{hadd mxC, forall m n, param (RmxA ==> RmxA ==> RmxA)
         (@addmx A m n) (@hadd_op _ _ _ m n)}.
```

We also have to prove the parametricity lemma associated to our recursion scheme on `positive`:

```
Instance param_elim_positive P P'
  (R : forall p, P p -> P' p -> Prop) txI txI' txO txO' txH txH' :
  (forall p, param (R p ==> R (p~1)) (txI p) (txI' p)) ->
  (forall p, param (R p ==> R (p~0)) (txO p) (txO' p)) ->
  (param (R 1) txH txH') ->
  forall p, param (R p) (positive_rect P txI txO txH p)
                        (positive_rect P' txI' txO' txH' p).
```

We declare this lemma as an `Instance` of the `param` type class. This allows to automate data refinement proofs requiring induction over `positive`. Finally, we prove parametricity lemmas for `Strassen_step` and `Strassen`:

```
Instance param_Strassen_step p :
   param (RmxA ==> RmxA ==> (RmxA ==> RmxA ==> RmxA) ==> RmxA)
         (@Strassen_step (@matrix A) p) (@Strassen_step mxC p).

Instance param_Strassen p :
   param (RmxA ==> RmxA ==> RmxA)
         (@Strassen (@matrix A) p) (@Strassen mxC p).
```

Here, the improvement over [8] is twofold: only one generic implementation of the algorithm is now required and refinement proofs are now mostly automated, including induction steps.

A possible drawback is that our generic description of the algorithms requires all the operators to take the sizes of the matrices involved as arguments, which are sometimes not required for computation-oriented operators. However, some preliminary benchmarks seem to indicate that this does not entail a significant performance penalty.

6 Related Work

Our work addresses a fundamental problem: how to change data representations in a compositional way. As such, it is no surprise that it shares aims with other work. We already mentioned ML-like modules and functors, that are available in Coq, but forbid proof methods to have a computational content.

The most general example of refinement relations we consider are partial quotients, which are often represented in type theory by setoids over partial equivalence relations [2] and manipulated using generalized rewriting [22]. The techniques we are using are very close to a kind of heterogeneous version of the latter. Indeed, it usually involves a relation `R : A -> A -> Prop` for a given type `A`, whereas our refinement relations have the shape `R : A -> B -> Prop` where `A` and `B` can be two different types.

Some years ago, a plugin was developed for Coq for changing data representations and converting proofs from a type to another [16]. However, this approach was limited to isomorphic types, and does not provide a way to achieve generic programming (only proofs are ported). Our design is thus more general, and we do not rely on an external plugin which can be costly to maintain.

In [15], a methodology for modular specification and development of programs in type theory is presented. The key idea is to express algebraic specifications using sigma-types which can be refined using refinement maps, and realized by concrete programs. This approach is close to the use of ML-like modules, since objects are abstracted and their behavior is represented by a set of equational properties. A key difference to our work is that these equational properties are stated using an abstract congruence relation, while we aim at proving correctness on objects that can be compared with Leibniz equality, making reasoning more convenient. This is made possible by our more relaxed relation between proof-oriented and computation-oriented representations.

Another way to reconcile data abstraction and computational content is the use of *views* [17,26]. In particular, it allows to derive induction schemes independently of concrete representations of data. This can be used in our setting to write generic programs utilizing these induction schemes for defining recursive programs and proving properties for generic types, in particular `param_elim_positive` (Sect. 5) is an example of a view.

The closest work to ours is probably the automatic data refinement tool Autoref implemented independently for Isabelle [14]. While many ideas, like the use of parametricity, are close to ours, the choice is made to rely on an external tool to synthesize executable instances of generic algorithms and refinement proofs. The richer formalism that we have at our disposal, in particular full poly-

morphism and dependent types makes it easier to internalize the instantiation of generic programs.

Another recent work that is related to this paper is [11] in which the authors explain how the Isabelle/HOL code generator uses data refinements to generate executable versions of abstract programs on abstract types like sets. In the paper they use a refinement relation that is very similar to our partial quotients (they use a domain predicate instead of an option type to denote what values are valid and which are not). The main difference though is that they are applying data refinements for code generation while in our case this is not necessary since all programs written in Coq can be executed as they are and data refinements are only useful to perform more efficient computations.

7 Conclusions and Future Work

In this paper an approach to data refinements has been presented where the user only needs to supply the minimum amount of necessary information and both programs and their correctness proofs gets transported to new data representation. The three main parts of the approach are:

1. a lightweight and general refinement interface to support any heterogeneous relation between two types,
2. operational type classes to increase generality of implementations and
3. parametricity to automatically transport correctness proofs.

As mentioned in the introduction of this paper, this work is an improvement of a previous work [8]. More precisely it improves the approach presented in Sect. 5 of [8] in the following aspects.

1. Generality: it extends to previously unsupported data types, like the type of non-normalized rationals (Sect. 2.2).
2. Modularity: each operator is refined in isolation instead of refining whole algebraic structures (Sect. 2.3), as suggested in the future work section of the previous paper.
3. Genericity: before, every operation had to be implemented both for the proof-oriented and computation-oriented types, now only one generic implementation is sufficient (Sect. 3).
4. Automation: the current approach has a clearer separation between the different steps of data refinements which makes it possible to use parametricity (Sect. 4) in order to automate proofs that previously had to be done by hand.

The implementation of points 2, 3 and 4 relies on the type class mechanism of Coq in two different ways: in order to support ad-hoc polymorphism of algebraic operations, and in order to do proof and term reconstruction automatically through logic programming. The automation of proof and term search is achieved by the same set of lemmas as in the previous paper, but now these do not impact the interesting proofs anymore.

The use of operational type classes is very convenient for generic programming. But the more complicated programs get, the more arguments they need. In particular, we may want to bundle operators in order to reduce the size of contexts that users need to write when defining generic algorithms.

The handling of parametricity is currently done by meta-programming but requires some user input and deals only with polymorphic constructions. We should address these two issues by providing a systematic way of producing parametricity lemmas for inductive types [3] and extending relation constructions with dependent types. We may adopt Keller and Lasson's [13] way of producing parametricity theorems and their proofs for closed terms.

Currently all formalizations have been done using standard Coq, but it would be interesting to see how the univalent foundations [18] can be used for simplifying our approach to data refinements. Indeed, in the presence of the univalence axiom, isomorphic structures are equal [1,7] which should be useful when refining isomorphic types. Also in the univalent foundations there are ways to represent quotient types (see for example [20]). This could be used to refine types that are related by quotients or even partial quotients.

The work presented in this paper is currently being used as a new basis for CoqEAL — The Coq Effective Algebra Library — which is a library, currently in development, containing many formally verified program refinements, for instance: Strassen's fast matrix product [25], Karatsuba's fast polynomial product [12], the Sasaki-Murao algorithm for efficiently computing the characteristic polynomial of a matrix [21] and an algorithm for computing the Smith normal form of matrices over Euclidean rings.

Acknowledgments: The authors are grateful to the anonymous reviewers for their useful comments and feedback. We also thank Bassel Mannaa and Dan Rosén for proof reading the final version of this paper.

References

1. B. Ahrens, C. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion, 2013. Preprint. `http://arxiv.org/abs/1303.0584`.
2. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.
3. J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free. *Journal of Functional Programming*, 22:107–152, 2 2012.
4. J. Chrzaszcz. Implementing Modules in the Coq System. In *TPHOLs*, volume 2758 of *LNCS*, pages 270–286. Springer, 2003.
5. C. Cohen. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 213–228, 2013.
6. Coq development team. The Coq Proof Assistant Reference Manual, version 8.4. Technical report, Inria, 2012.
7. N. A. Danielsson and T. Coquand. Isomorphism is Equality, 2013. Preprint. `http://www.cse.chalmers.se/~nad/publications/coquand-danielsson-isomorphism-is-equality.html`.

8. M. Dénès, A. Mörtberg, and V. Siles. A Refinement Based Approach to Computational Algebra in Coq. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 83–98, 2012.
9. G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009.
10. B. Gregoire and A. Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *TPHOLs*, LNCS, pages 98–113. Springer, 2005.
11. F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data Refinement in Isabelle/HOL. In *Interactive Theorem Proving*, LNCS. Springer, 2013.
12. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *USSR Academy of Sciences*, volume 145, pages 293–294, 1962.
13. C. Keller and M. Lasson. Parametricity in an Impredicative Sort. In *CSL*, volume 16, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
14. P. Lammich. Automatic Data Refinement. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 84–99, 2013.
15. Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.
16. N. Magaud. Changing Data Representation within the Coq System. In *TPHOLs*, volume 2758 of *LNCS*, pages 87–102. Springer, 2003.
17. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
18. T. U. F. Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. `http://homotopytypetheory.org/book/`.
19. J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
20. E. Rijke and B. Spitters. Sets in homotopy type theory, 2013. Preprint. `http://arxiv.org/abs/1305.3835`.
21. T. Sasaki and H. Murao. Efficient Gaussian Elimination Method for Symbolic Determinants and Linear Systems. *ACM Trans. Math. Softw.*, 8(3):277–289, Sept. 1982.
22. M. Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.
23. M. Sozeau and N. Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.
24. B. Spitters and E. van der Weegen. Type Classes for Mathematics in Type Theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.
25. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, Aug. 1969.
26. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, pages 307–313. ACM Press, 1987.
27. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.